Sytare: Persistence de l'état des périphériques pour les systèmes à alimentation intermittente

Gautier Berthou, Tristan Delizy, Kevin Marquet, Guillaume Salagnac, Tanguy Risset

Compas'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Sytare: Persistence de l'état des périphériques pour les systèmes à alimentation intermittente

Gautier Berthou, Tristan Delizy, Kevin Marquet, Guillaume Salagnac, Tanguy Risset

Univ Lyon, INSA Lyon, Inria, CITI,
F-69621 Villeurbanne, France.
firstname.lastname@inria.fr

Résumé
Les systèmes dits *à alimentation intermittente* sont de petits systèmes embarqués récupérant l'énergie dans leur environnement. À cause de contraintes de taille et de coût, ils subissent de fréquentes coupures de courant, mais sont néanmoins capables d'exécuter un programme logiciel, en sauvegardant les données nécessaires au calcul dans une mémoire non-volatile. Cet article présente une technique permettant à ces systèmes d'utiliser des périphériques non triviaux tels qu'un convertisseur analogique-numérique, une interface série ou une radio.

Mots-clés : système embarqué, alimentation intermittente, périphériques matériels, gestion mémoire, checkpointing, système d'exploitation

1. Introduction

Les progrès technologiques permettent petit à petit d'embarquer par centaines de tout petits systèmes presque partout, rendant ainsi des services variés dans différents domaines tels que la santé, les transports ou les loisirs. Les plus petits de ces systèmes utilisent un processeur lent (fréquence de l'ordre du MHz), peu de mémoire (quelques dizaines de kilo-octets), et des périphériques simples. Mais alimenter de tels systèmes est un problème car leur taille, leur prix, et le peu de maintenance qu'il peut leur être fourni interdisent l'usage d'une batterie.
Récupérer l'énergie dans l'environnement est un moyen prometteur de résoudre ce problème. À cause de leur taille, de leur prix, et souvent de la faible luminosité auquel est exposé le système, l'usage de panneaux solaires est interdit, mais d'autres moyens existent : récupération par ondes radio, par gradient de température.... Cela permet l'apparition progressive de systèmes à alimentation intermittente (TPS — *Transiently-Powered Systems*), qui amassent l'énergie dans un condensateur et l'utilisent par cycles de quelques dizaines de millisecondes. Ces systèmes sont donc sujets à des coupures très fréquentes et imprédictibles. Ces coupures rendent impossible la programmation traditionnelle à base de processus s'exécutant sur des systèmes tels que FreeRTOS [6] ou Contiki [5]. Ces derniers ne supportent pas les fréquentes coupures de courant, ils doivent décider quand s'arrêter et quand redémarrer.
La récente apparition de mémoires non-volatiles rapides (*NVRAM — Non-Volatile RAM*) permet d'avancer vers des systèmes conservant leur état même lorsque leur exécution est entrecoupée de coupures de courant. Plusieurs travaux récents ont proposé de sauvegarder l'état du système avant une coupure en utilisant des mécanismes de *checkpointing*. Si ces travaux permettent la savegarde et la restauration de l'état de la partie *calcul* du système (registres du

processeur, pile d'exécution, variables globales), ils ne considèrent que des applications simples n'utilisant pas de périphériques complexes tels qu'un ADC, un contrôleur de bus ou un emetteur/récepteur radio. Ces périphériques ont pourtant un état à sauvegarder.
Cet article propose une nouvelle technique de checkpointing pour systèmes à alimentation intermittente incluant de la RAM non-volatile. Les mécanismes mis en œuvre permettent l'usage correct de périphériques complexes en présence de coupures de courant.

2. Contexte et problématique

Cette partie détaille la problématique de cet article : comment permettre aux TPS d'utiliser des périphériques non triviaux.

2.1. Les systèmes à alimentation intermittentes (TPS)

Les systèmes embarqués tels que les téléphones ou les nœuds de réseaux de capteurs fonctionnent tous sur batterie. Dans certaines situations il est cependant impossible d'utiliser une batterie [7]. Dans ces cas là, le système doit récupérer l'énergie dans son environnement, de sources telles qu'un piezo électrique, un gradient de température ou d'ondes radio [11]. Les cartes à puce et les *RFID* sont des exemples connus, mais d'autres plates-formes existent. Par exemple, la WISP (*Intel's Wireless Identification and Sensing Platform*) d'Intel, ou encore la M^3 [9] mesurant $1.0mm^3$, récupérant l'énergie dans l'environnement, et embarquant un Cortex-M0, quelques kilo-octets de SRAM et quelques kilo-octets de NVRAM.
Une caractéristique commune de ces systèmes est qu'ils doivent gérer une alimentation imprédictible. Même si de l'énergie est disponible, le niveau d'énergie récupérée est très bas [11] en comparaison de ce que le système consomme quand il est actif. Par exemple, les cartes à puce sans contact doivent effectuer un transaction complète en quelques centaines de millisecondes. Au-delà de la faisabilité, les contraintes imposées au programmeur de ces systèmes sont très dures. Il est important de fournir à celui-ci un système d'exploitation séparé des applications, abstrayant pour lui la plus grande partie de ces contraintes.

2.2. TPS et NVRAM

Un problème évident de l'alimentation intermittente est que le système va perdre ses données volatiles lors de chaque panne. Dans un TPS, l'état du processeur et le contenu de la RAM sont perdus, aussi bien que l'état des périphériques. Ces dernières années, des avancées ont été faites dans le domaine des mémoires non-volatiles. Certaines types NVRAM promettent de supprimer à terme la distinction entre mémoire non-volatile, lente, de *stockage*, et mémoire rapide, volatile, de *travail* [1]. Dans un TPS, remplacer naïvement la RAM par de la NVRAM a des effets indésirables. Puisque les pannes sont fréquentes, elles peuvent arriver alors qu'une structure de données non-volatile est en train d'être modifiée. Lorsque la plate-forme redémarrera, le programme reprendra avec un état incohérent, provoquant le problème dit de la *broken time machine* [12].
Une solution possible est d'avoir un processeur non-volatile [10]. Par exemple, Bartling et al. [3] ont mis au point un tel micro-contrôleur non-volatile. Ce type d'approche est intéressant en termes d'architecture mais a des limitations majeures en termes de modèle de programmation. De plus, conserver une structure de données en NVRAM la rend persistente mais nécessite aussi que chaque accès peut être lent ou coûteux en énergie, en fonction de la technologie sous-jacente. D'un autre côté, stocker en RAM permet de bonnes performances mais pose le problème de la volatilité des données. Pour ces raisons, la plupart des systèmes utilisent maintenant une combinaison de RAM et NVRAM [3].

Afin de conserver la cohérence entre les données en NVRAM et les données en RAM, une solution très utilisée est le *checkpointing* : les coupures de courant sont détectées et avant l'arrêt complet en résultant, toutes les données volatiles nécessaires sont sauvegardées en NVRAM. Quand la plate-forme redémarre, les données sont restaurées afin que le programme reprenne son exécution d'une manière transparente.

2.3. Checkpointing

Sauvegarder l'état des données en NVRAM nécessite la détection des pannes de courant. Cela est souvent implémenté par un pont diviseur de tension, en utilisant un comparateur de tension intégré au microcontrolleur (voir [2] par exemple).

Dans presque toutes les techniques de checkpointing, deux structures sont utilisées : une image valide des données volatiles — que nous appelons *dernier checkpoint* —, et le *prochain checkpoint* (pas encore valide) en construction. Si une panne survient, toutes les données volatiles de l'exécution en cours seront copiées vers le checkpoint en construction (le *prochain*) et si la copie se termine correctement les pointeurs *prochain* et *dernier* seront inversés. Au redémarrage de la plate-forme, le *dernier* checkpoint sera restauré.

Cependant, ces travaux se concentrent sur les calculs et proposent de sauvegarder les registres du CPU, la pile d'exécution et les variables gloabales : ils ignorent les périphériques. Or, les péripériques non-triviaux tels que les ADCs, les liaisons série, ou les radios, ont une machine à états contrôlant leur exécution et qui ne peut être restaurée simplement. L'état interne du périphérique est le plus souvent accessible à travers un pilote de périphérique (que nous appellerons *driver* par la suite).

Dans la suite de l'article, nous expliquons comment rendre l'état des périphériques persistant (résoudre le *problème de la volatilité de l'état des périphériques*), et cohérent au fil des redémarrages de manière à ce que l'application soit insensible aux pannes de courant (résoudre le *problème de l'atomicité des accès aux périphériques*).

3. Checkpointing des périphériques

Nos mécanismes sont implémentés dans une couche logicielle entre code applicatif et code driver. Nous appelons cette couche logicielle le *noyau*. De cette manière, nous les rendons plus transparents pour le programmeur. Cette solution fait l'hypothèse que l'état des périphériques ne peut changer que suite à l'exécution de code driver.

Le code noyau est donc en charge de sauvegarder et restaurer l'état des périphériques et des drivers. Ces états sont stockés en NVRAM dans une structure de données que nous appelons *device context*, contenant les informations nécessaires au périphérique pour restaurer son état. Mais le noyau ne *committera* en NVRAM les changements effectués sur un *device context* que lorsque le péripéhérique associé aura atteint un état stable. Nous détaillons légèrement ce mécanisme ci-après mais pour plus de détails, veuillez vous référer au rappport de recherche [4].

3.1. Non-volatilité de l'état des périphériques

Restaurer l'état d'un périphérique nécessite des opérations non triviales telles que configurer certaines broches, respecter certains délais etc. Dans notre proposition, une fonction de restauration est responsable de l'initialisation d'un périphérique et de le ramener dans l'état décrit dans son *device context*. Pour les périphériques complexes nécessitant des accès indirects (via bus SPI par exemple) ou imposant certaines contraintes particulières (délais, séquence d'instructions particulières…), cette fonction peut être complexe.

3.2. Atomicité des accès aux périphériques

Restaurer le contexte d'un périphérique est différent de restaurer un contexte applicatif : un point d'exécution dans un code *driver* ne reflète pas l'état courant d'un périphérique. Les appels

au *driver* doivent être protégés. Chaque fois qu'un tel appel survient, il est d'abord pris en charge par le noyau, nous appelons cet appel au noyau *appel système (syscall)* par analogie avec le concept homonyme dans les systèmes d'exploitation classiques.
Le contrat entre l'application et le noyau spécifie qu'un syscall sera entièrement exécuté en un cycle de vie de la plate-forme. Si une panne survient au milieu de l'appel d'une fonction du *driver*, alors au prochain démarrage, cette fonction sera re-exécutée depuis le début (au lieu de juste reprendre où elle avait été interrompue). Cela implique que le checkpointing du contexte applicatif et du contexte des *drivers* sont gérés différemment.
Nous isolons donc le flot de contrôle des drivers et le flot de control applicatif en exécutant les appels aux pilotes dans une *pile d'exécution séparée*. Cette pile, appelée *pile noyau*, n'est jamais sauvegardée dans un checkpoint. Cela garantit qu'un progrès partiel au sein d'un code *driver* sera volatile. Cette gestion de pile est effectuée par le noyau au début et à la fin d'un syscall.
Plusieurs détails d'importance variée sont à prendre en compte — par exemple lorsqu'un driver en appelle un autre —, le lecteur intéressé pourra lire le rapport de recherche [4].

3.3. Illustration

Le scénario simple décrit par la figure 1 illustre le mécanisme implémenté. Initialement, l'état de l'application est *App_0* et le périphérique *A* a un état *A_0*. Le programme utilisateur demande l'accès au périphérique *A*, ce qui doit être effectué via l'API Sytare (*i.e.* la colonne *kernel* à gauche de la figure).
Le *last checkpoint* sur la droite du schéma représente l'état qui sera restauré si une panne survient et que le système n'a pas le temps de sauvegarder cet état. Le *Next checkpoint* représente le checkpoint en cours de construction. Dans ces checkpoints, la colonne *App* indique l'état de l'application à restaurer, et la colonne *driver* indique l'état du driver *A* à restaurer. La colonne *current driver call* indique la sauvegarde de l'appel au driver en cours d'exécution.
Le diagramme de séquence sur la gauche décrit un appel de la fonction du driver `drvA_fn()` depuis le programme utilisateur. Il montre les syscalls du noyau (`syt_drvA_fn()`) et les interactions entre le noyau et le driver. Après que le syscall a été effectué correctement, une panne est détectée, et un checkpointing est effectué. Juste avant la panne complète, le noyau fait pointer le *last checkpoint* sur le *next checkpoint*. L'état du programme est donc le suivant : l'application est dans l'état *App_2*, le driver *A* est dans l'état *A_1*, aucun driver n'a été interrompu.
La primitive `syt_signal()` est utilisée pour notifier le noyau que le périphérique *A* a changé d'état. La promitive `syt_commit()` rend persistent le nouvel état du périphérique dans le *Next checkpoint* et supprime *A* de la liste des périphériques modifiés.

4. Validation expérimentale

4.1. Platforme matérielle

Notre prototype est implémenté au-dessus de la carte d'évaluation Texas Instruments MSP-EXP430FR5739 [1]. Le microcontrôleur MSP430FR5739 inclut 16ko de FRAM (i.e. NVRAM ferroélectrique) et 1ko de RAM. La limitation principale de la FRAM est son temps d'accès de 125ns, limitant la fréquence d'accès à cette mémoire à 8MHz. Le reste de la plate-forme (CPU, RAM, périphériques) peut être cadencé à 24MHz, donc nos expériences sont faites à cette fréquence. Nous ajoutons à cette plate-forme une carte fille comportant un émetteur-récepteur radio CC2500 [2], connecté au microcontrôleur par un bus SPI.

1. `http://www.ti.com/lit/ug/slau343b/slau343b.pdf`
2. `http://www.ti.com/lit/ds/swrs040c/swrs040c.pdf`

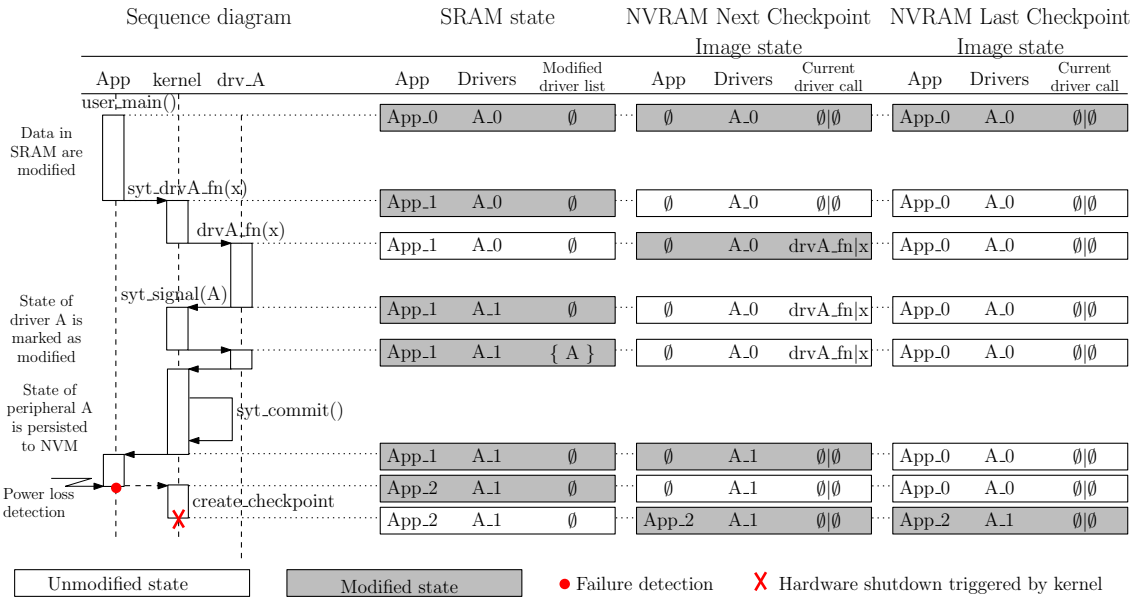

FIGURE 1 – Diagramme de séquence d'un simple syscall, montrant le contenu des structures du noyau en RAM et en NVRAM.

4.2. Alimentation et détection des pannes

Dans un souci de reproductibilité, nous alimentons le système avec un générateur de tension programmable. Le signal est directement connecté aux broches *Vcc* et *Gnd* de la carte.

Le microcontrôleur msp430 comporte un comparateur de tension que nous utilisons pour détecter les pannes de courant. Comme dans les travaux comparables de l'état de l'art [2, 8] nous ajoutons un pont diviseur de tension composé de deux résistances de 1MΩ chacune, pour que le comparateur mesure une tension différente de sa propre tension de fonctionnement. La consommation supplémentaire due à ce montage (1.6µA) est négligeable devant la consommation de la plate-forme (qui peut atteindre plusieurs mA).

4.3. Applications témoin

Nous avons évalué Sytare à l'aide de quatre applications utilisant des périphériques matériels plus ou moins complexes. «*RSA*» fait du chiffrement. Elle n'utilise pas de périphérique mais a une empreinte mémoire significative. «*LEDs*» affiche un simple compteur logiciel sur les diodes de la plate-forme. «*ADC Sense*» mesure la température ambiante avec le convertisseur analogique-numérique et stocke les valeurs dans un tableau. Enfin, «*WSN*» mesure la température et transmet les valeurs par radio vers un récepteur (qui est alimenté continûment).

Le choix de ces applications couvre plusieurs types de périphériques : simples comme les diodes, ou plus complexes comme la puce radio qui est accessible au travers d'un bus SPI.

4.4. Évaluation des performances

Pour chaque application, on mesure T_{wired} comme le temps mis par l'application «pure» pour s'exécuter en entier, sous alimentation continue et sans le noyau Sytare. Pour les applications périodiques, le point d'arrivée est fixé arbitrairement, par exemple envoyer 10 messages, de façon à ce que la durée d'exécution soit assez grande.

Par contraste en mode intermittent, on note T_{on} la durée pendant laquelle la plate-forme est alimentée entre deux périodes d'extinction. Pour un T_{on} donné, nous définissons $T_{transient}(T_{on})$ comme le temps requis par le système pour exécuter l'application jusqu'au point d'arrivée. Les

durées $T_{transient}$ (et T_{wired}) n'incluent pas les moments où la plate-forme est éteinte, mais par contre le temps de démarrage du matériel est toujours comptabilisé.
On définit ensuite le rendement $Y(T_{on})$ du système comme le rapport $\frac{T_{wired}}{T_{transient}(T_{on})}$. La figure 2 illustre le rendement obtenu sur nos différentes applications témoin en fonction de T_{on}.

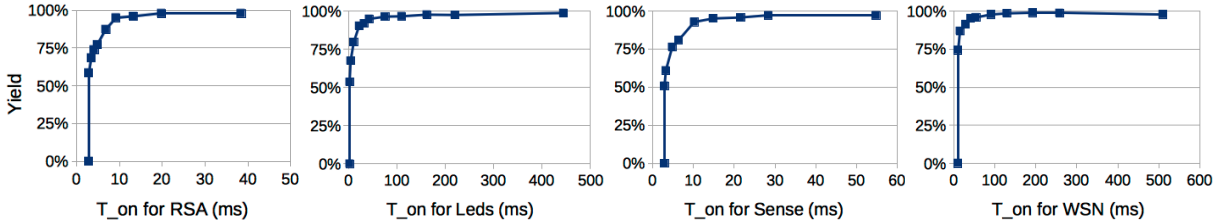

FIGURE 2 – Rendement Y obtenu pour une durée T_{on} des périodes d'alimentation de 0 à T_{wired}.

Pour des T_{on} trop courtes, le système n'a jamais assez de temps entre deux coupures pour successivement démarrer, faire progresser l'application, et réussir un checkpoint. La métrique $T_{transient}$ est alors «infinie» et donc Y est 0. On mesure ainsi T_{on}^{min} comme la plus petite durée d'alimentation T_{on} qui produise un rendement non-nul.
À l'inverse, quand T_{on} approche T_{wired}, alors l'application pourra s'exécuter en entier avant la première coupure. Mais Y n'atteindra jamais 100% à cause du temps passé à exécuter du code noyau (initialisation et syscalls). On note Y^{max} le meilleur rendement ainsi observé. La figure 3 donne les valeurs de ces deux métriques globales pour nos différentes applications.
Le surcoût occasionné par Sytare sur le temps d'exécution des appels drivers est très différent d'un driver à l'autre. La figure 4 donne une évaluation de ces surcoûts.

	RSA	LEDs	Sense	WSN
T_{on}^{min}	2.79ms	2.79ms	2.90ms	9.40ms
Y^{max}	0.98	0.99	0.97	0.99

FIGURE 3 – Performances globales : durée d'alimentation minimum supportable, et rendement maximum atteignable.

Hardware action	Time overhead
ADC sample	+27%
Radio sleep	+137%
Radio wake up	+8%
Radio message send	+1%

FIGURE 4 – Appels drivers : surcoût (en temps) occasionné par le noyau.

5. Conclusion

Cet article présente Sytare, une couche logicielle facilitant le développement d'applications embarquées sur des plates-formes à alimentation intermittente. L'exécution de l'application est étalée automatiquement sur plusieurs périodes d'alimentation, sans intervention de la part du programmeur. La contribution principale est un mécanisme garantissant l'accès correct aux périphériques matériels malgré les coupures de courant. Les plates-formes visées sont par exemple les cartes à puces sans-contact, ou les RFID programmables, et plus généralement les petits systèmes embarqués destinés à l'Internet des objets.
Ce travail ouvre de nombreuses perspectives. Dans un premier temps, nous comptons développer des drivers pour davantage de périphériques, afin de mieux cerner les limitations imposées par nos hypothèses de travail. Aussi, implémenter des applications supplémentaires nous permettrait de mieux comprendre comment l'intermitence modifie la sémantique offerte au programmeur. Dans une optique à plus long terme, nous souhaitons travailler à l'intégration de Sytare dans un système d'exploitation embarqué comme Contiki ou Riot.

Bibliographie

1. Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *HotOS 2011 : 13th USENIX conference on Hot topics in Operating Systems*, 2011.
2. Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus : Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1), 2015.
3. Steven Bartling, Sudhanshu Khanna, Michael Clinton, Scott R. Summerfelt, John A. Rodriguez, and Hugh P. McAdams. An 8mhz 75ua/mhz zero-leakage non-volatile logic-based cortex-m0 mcu soc exhibiting 100-percent digital state retention at vdd=0v with <400ns wakeup and sleep transitions. In *ISSCC 2013 : IEEE International Solid-State Circuits Conference*, pages 432–433, 2013.
4. Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. Peripheral State Persistence For Transiently Powered Systems. Research Report 9018, INRIA, February 2017.
5. Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki : a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*. IEEE, 2004.
6. Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open source freertos as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118, 2016.
7. Hrishikesh Jayakumar, Kangwoo Lee, Woo Suk Lee, Arnab Raha, Younghyun Kim, and Vijay Raghunathan. Powering the internet of things. In *ISPLED'14 : International Symposium on Low Power Electronics and Design*, 2014.
8. Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. QUICKRECALL : A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *VLSID 2014 : 27th International IEEE Conference on VLSI Design and 13th International Conference on Embedded Systems*, pages 330–335, 2014.
9. Y. Lee, S. Bang, I. Lee, Y. Kim, G. Kim, M. H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw. A modular 1 mm^3 die-stacked sensing platform with low power I2C inter-die communication and multi-modal energy harvesting. *IEEE Journal of Solid-State Circuits*, 48(1), 2013.
10. Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, Jiwu Shu, and Huazhong Yang. Ambient energy harvesting nonvolatile processors : from circuit to system. In *DAC 2015 : 52nd Annual Design Automation Conference*, pages 150 :1–150 :6, 2015.
11. Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *HPCA'15 : High Performance Computer Architecture*, pages 526–537. IEEE, 2015.
12. Benjamin Ransford and Brandon Lucia. Nonvolatile memory is a broken time machine. In *MSPC 2014 : ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2014.